CS 6210: Matrix Computations

Gaussian elimination

David Bindel

2025-09-15

Introduction

For the next few lectures, we will explore methods to solve linear systems. Our main tool will be the factorization $PA = LU$, where P is a permutation, L is a unit lower triangular matrix, and U is an upper triangular matrix. As we will see, the Gaussian elimination algorithm learned in a first linear algebra class implicitly computes this decomposition; but by thinking about the decomposition explicitly, we can come up with other organizations for the computation.

We emphasize a few points up front:

- *Some matrices are singular.* Errors in this part of the class often involve attempting to invert a matrix that has no inverse. A matrix does *not* have to be invertible to admit an LU factorization. We will also see more subtle problems from *almost* singular matrices.
- *Some matrices are rectangular.* In this part of the class, we will deal almost exclusively with square matrices; if a rectangular matrix shows up, we will try to be explicit about dimensions. That said, LU factorization makes sense for rectangular matrices as well as for square matrices — and it is sometimes useful.
- *`inv` is evil.* The `inv` command is one of the most abused commands in . The backslash operator is the preferred way to solve a linear system absent other information:

  ```
  x = A \ b        # Good
  x = inv(A) * b   # Evil
  ```

  Homework solutions that feature inappropriate explicit `inv` commands *will* lose points.
- *LU is not for linear solves alone.* One can solve a variety of other interesting problems with an LU factorization.
- *LU is not the only way to solve systems.* Gaussian elimination and variants will be our default solver, but there are other solver methods that are appropriate for problems with more structure. We will touch on other methods throughout the class.

Gaussian elimination by example

Let's start our discussion of LU factorization by working through these ideas with a concrete example:

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{bmatrix}.$$

To eliminate the subdiagonal entries a_{21} and a_{31}, we subtract twice the first row from the second row, and thrice the first row from the third row:

$$A^{(1)} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{bmatrix} - \begin{bmatrix} 0\cdot 1 & 0\cdot 4 & 0\cdot 7 \\ 2\cdot 1 & 2\cdot 4 & 2\cdot 7 \\ 3\cdot 1 & 3\cdot 4 & 3\cdot 7 \end{bmatrix} = \begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{bmatrix}.$$

That is, the step comes from a rank-1 update to the matrix:

$$A^{(1)} = A - \begin{bmatrix} 0 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \end{bmatrix}.$$

Another way to think of this step is as a linear transformation $A^{(1)} = M_1 A$, where the rows of M_1 describe the multiples of rows of the original matrix that go into rows of the updated matrix:

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} = I - \begin{bmatrix} 0 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} = I - \tau_1 e_1^T.$$

Similarly, in the second step of the algorithm, we subtract twice the second row from the third row:

$$\begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{bmatrix} = \left(I - \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \right) A^{(1)}.$$

More compactly: $U = (I - \tau_2 e_2^T) A^{(1)}$.

Putting everything together, we have computed

$$U = (I - \tau_2 e_2^T)(I - \tau_1 e_1^T) A.$$

Therefore,

$$A = (I - \tau_1 e_1^T)^{-1} (I - \tau_2 e_2^T)^{-1} U = LU.$$

Now, note that

$$(I - \tau_1 e_1^T)(I + \tau_1 e_1^T) = I - \tau_1 e_1^T + \tau_1 e_1^T - \tau_1 e_1^T \tau_1 e_1^T = I,$$

since $e_1^T\tau_1$ (the first entry of τ_1) is zero. Therefore,

$$(I - \tau_1 e_1^T)^{-1} = (I + \tau_1 e_1^T)$$

Similarly,

$$(I - \tau_2 e_2^T)^{-1} = (I + \tau_2 e_2^T)$$

Thus,

$$L = (I + \tau_1 e_1^T)(I + \tau_2 e_2^T).$$

Now, note that because τ_2 is only nonzero in the third element, $e_1^T\tau_2 = 0$; thus,

$$\begin{aligned} L &= (I + \tau_1 e_1^T)(I + \tau_2 e_2^T) \\ &= (I + \tau_1 e_1^T + \tau_2 e_2^T + \tau_1 (e_1^T \tau_2) e_2^T \\ &= I + \tau_1 e_1^T + \tau_2 e_2^T \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix}. \end{aligned}$$

The final factorization is

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix} = LU.$$

Note that the subdiagonal elements of L are easy to read off: for $j > i$, l_{ij} is the multiple of row j that we subtract from row i during elimination. This means that it is easy to read off the subdiagonal entries of L during the elimination process.

Basic LU factorization

Let's generalize our previous algorithm and write a simple code for LU factorization. We will leave the issue of pivoting to a later discussion. We'll start with a purely loop-based implementation:

```
#
# Compute LU factors in separate storage
#
function mylu_v1(A)
    n = size(A)[1]
    L = Matrix(1.0I, n, n)
    U = copy(A)
    for j = 1:n-1
```

```
        for i = j+1:n

            # Figure out multiple of row j to subtract from row i
            L[i,j] = U[i,j]/U[j,j]

            # Subtract off the appropriate multiple of row j from row i
            U[i,j] = 0.0
            for k = j+1:n
                U[i,k] -= L[i,j]*U[j,k]
            end
        end
    end
    L, U
end
```

Note that we can write the two innermost loops more concisely by thinking of them in terms of applying a *Gauss transformation* $M_j = I - \tau_j e_j^T$, where τ_j is the vector of multipliers that appear when eliminating in column j. Also, note that in LU factorization, the locations where we write the multipliers in L are exactly the same locations where we introduce zeros in A as we transform to U. Thus, we can re-use the storage space for A to store both L (except for the diagonal ones, which are implicit) and U. Using this strategy, we have the following code:

```
#
# Compute LU factors in packed storage
#
function mylu_v2(A)
    n = size(A)[1]
    A = copy(A)
    for j = 1:n-1
        A[j+1:n,j] /= A[j,j]                      # Form vector of multipliers
        A[j+1:n,j+1:n] -= A[j+1:n,j]*A[j,j+1:n]'  # Update Schur complement
    end
    UnitLowerTriangular(A), UpperTriangular(A)
end
```

The bulk of the work at step j of the elimination algorithm is in the computation of a rank-one update to the trailing submatrix. How much work is there in total? In eliminating column j, we do $(n-j)^2$ multiplies and the same number of subtractions; so in all, the number of multiplies (and adds) is

$$\sum_{j=1}^{n-1} (n-j)^2 = \sum_{k=1}^{n-1} k^2 = \frac{1}{3}n^3 + O(n^2)$$

We also perform $O(n^2)$ divisions. Thus, Gaussian elimination, like matrix multiplication, is an $O(n^3)$ algorithm operating on $O(n^2)$ data.

Schur complements

The idea of expressing a step of Gaussian elimination as a low-rank submatrix update turns out to be sufficiently useful that we give it a name. At any given step of Gaussian elimination, the trailing submatrix is called a *Schur complement*. We investigate the structure of the Schur complements by looking at an LU factorization in block 2-by-2 form:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{22}U_{22} + L_{21}U_{12} \end{bmatrix}.$$

We can compute L_{11} and U_{11} as LU factors of the leading sub-block A_{11}, and

$$\begin{aligned} U_{12} &= L_{11}^{-1} A_{12} \\ L_{21} &= A_{21} U_{11}^{-1}. \end{aligned}$$

What about L_{22} and U_{22}? We have

$$\begin{aligned} L_{22}U_{22} &= A_{22} - L_{21}U_{12} \\ &= A_{22} - A_{21}U_{11}^{-1}L_{11}^{-1}A_{12} \\ &= A_{22} - A_{21}A_{11}^{-1}A_{12}. \end{aligned}$$

This matrix $S = A_{22} - A_{21}A_{11}^{-1}A_{12}$ is the block analogue of the rank-1 update computed in the first step of the standard Gaussian elimination algorithm.

For our purposes, the idea of a Schur complement is important because it will allow us to write blocked variants of Gaussian elimination — an idea we will take up in more detail shortly. But the Schur complement actually has meaning beyond being a matrix that mysteriously appears as a by-product of Gaussian elimination. In particular, note that if A and A_{11} are both invertible, then

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} X \\ S^{-1} \end{bmatrix} = \begin{bmatrix} 0 \\ I \end{bmatrix},$$

i.e. S^{-1} is the $(2,2)$ submatrix of A^{-1}.

Blocked Gaussian elimination

Just as we could rewrite matrix multiplication in block form, we can also rewrite Gaussian elimination in block form. For example, if we want

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

then we can write Gaussian elimination as:

1. Factor $A_{11} = L_{11}U_{11}$.
2. Compute $L_{21} = A_{21}U_{11}^{-1}$ and $U_{12} = L_{11}^{-1}A_{12}$.
3. Form the Schur complement $S = A_{22} - L_{21}U_{12}$ and factor $L_{22}U_{22} = S$.

This same idea works for more than a block 2-by-2 matrix. As with matrix multiply, thinking about Gaussian elimination in this blocky form lets us derive variants that have better cache efficiency. Notice that all the operations in this blocked code involve matrix-matrix multiplies and multiple back solves with the same matrix. These routines can be written in a cache-efficient way, since they do many floating point operations relative to the total amount of data involved.

Though some of you might make use of cache blocking ideas in your own work, most of you will never try to write a cache-efficient Gaussian elimination routine of your own. The routines in LAPACK and Julia (really the same routines) are plenty efficient, so you would most likely turn to them. Still, it is worth knowing how to think about block Gaussian elimination, because sometimes the ideas can be specialized to build fast solvers for linear systems when there are fast solvers for sub-matrices

For example, consider the *bordered* matrix

$$A = \begin{bmatrix} B & W \\ V^T & C \end{bmatrix},$$

where B is an n-by-n matrix for which we have a fast solver and C is a p-by-p matrix, $p \ll n$. We can factor A into a product of *block* lower and upper triangular factors with a simple form:

$$\begin{bmatrix} B & W \\ V^T & C \end{bmatrix} = \begin{bmatrix} B & 0 \\ V^T & L_{22} \end{bmatrix} \begin{bmatrix} I & B^{-1}W \\ 0 & U_{22} \end{bmatrix}$$

where $L_{22}U_{22} = C - V^T B^{-1} W$ is an ordinary (small) factorization of the trailing Schur complement. To solve the linear system

$$\begin{bmatrix} B & W \\ V^T & C \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix},$$

we would then run block forward and backward substitution:

$$\begin{aligned} y_1 &= B^{-1} b_1 \\ y_2 &= L_{22}^{-1} \left(b_2 - V^T y_1 \right) \end{aligned}$$

$$\begin{aligned} x_2 &= U_{22}^{-1} y_2 \\ x_1 &= y_1 - B^{-1}(W x_2) \end{aligned}$$

Backward error in Gaussian elimination

Solving $Ax = b$ in finite precision using Gaussian elimination followed by forward and backward substitution yields a computed solution $\hat{x}$ *exactly* satisfies

$$(A + \delta A)\hat{x} = b,$$

where $|\delta A| \lesssim 3n\epsilon_{\text{mach}}|\hat{L}||\hat{U}|$, assuming $\hat{L}$ and $\hat{U}$ are the computed L and U factors.

I will now briefly sketch a part of the error analysis following Demmel's treatment (§2.4.2). Mostly, this is because I find the treatment in §3.3.1 of Van Loan less clear than I would like – but also, the bound in Demmel's book is marginally tighter. Here is the idea. Suppose $\hat{L}$ and $\hat{U}$ are the computed L and U factors. We obtain $\hat{u}_{jk}$ by repeatedly subtracting $l_{ji}u_{ik}$ from the original a_{jk}, i.e.

$$\hat{u}_{jk} = \text{fl}\left(a_{jk} - \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}\right).$$

Regardless of the order of the sum, we get an error that looks like

$$\hat{u}_{jk} = a_{jk}(1+\delta_0) - \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}(1+\delta_i) + O(\epsilon_{\text{mach}}^2)$$

where $|\delta_i| \leq (j-1)\epsilon_{\text{mach}}$. Turning this around gives

$$\begin{aligned} a_{jk} &= \frac{1}{1+\delta_0}\left(\hat{l}_{jj}\hat{u}_{jk} + \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}(1+\delta_i)\right) + O(\epsilon_{\text{mach}}^2) \\ &= \hat{l}_{jj}\hat{u}_{jk}(1-\delta_0) + \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}(1+\delta_i-\delta_0) + O(\epsilon_{\text{mach}}^2) \\ &= \left(\hat{L}\hat{U}\right)_{jk} + e_{jk}, \end{aligned}$$

where

$$e_{jk} = -\hat{l}_{jj}\hat{u}_{jk}\delta_0 + \sum_{i=1}^{j-1} \hat{l}_{ji}\hat{u}_{ik}(\delta_i - \delta_0) + O(\epsilon_{\text{mach}}^2)$$

is bounded in magnitude by $(j-1)\epsilon_{\text{mach}}(|L||U|)_{jk} + O(\epsilon_{\text{mach}}^2)$[1]. A similar argument for the components of $\hat{L}$ yields

$$A = \hat{L}\hat{U} + E, \text{ where } |E| \leq n\epsilon_{\text{mach}}|\hat{L}||\hat{U}| + O(\epsilon_{\text{mach}}^2).$$

In addition to the backward error due to the computation of the LU factors, there is also backward error in the forward and backward substitution phases, which gives the overall bound ([gauss-bnd]).

[1] It's obvious that e_{jk} is bounded in magnitude by $2(j-1)\epsilon_{\text{mach}}(|L||U|)_{jk} + O(\epsilon_{\text{mach}}^2)$. We cut a factor of two if we go down to the level of looking at the individual rounding errors during the dot product, because some of those errors cancel.

Pivoting

The backward error analysis in the previous section is not completely satisfactory, since $|L||U|$ may be much larger than $|A|$, yielding a large backward error overall. For example, consider the matrix

$$A = \begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \delta^{-1} & 1 \end{bmatrix} \begin{bmatrix} \delta & 1 \\ 0 & 1-\delta^{-1} \end{bmatrix}.$$

If $0 < \delta \ll 1$ then $\|L\|_\infty \|U\|_\infty \approx \delta^{-2}$, even though $\|A\|_\infty \approx 2$. The problem is that we ended up subtracting a huge multiple of the first row from the second row because δ is close to zero — that is, the leading principle minor is *nearly* singular. If δ were exactly zero, then the factorization would fall apart even in exact arithmetic. The solution to the woes of singular and near singular minors is pivoting; instead of solving a system with A, we re-order the equations to get

$$\hat{A} = \begin{bmatrix} 1 & 1 \\ \delta & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \delta & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1-\delta \end{bmatrix}.$$

Now the triangular factors for the re-ordered system matrix $\hat{A}$ have very modest norms, and so we are happy. If we think of the re-ordering as the effect of a permutation matrix P, we can write

$$A = \begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \delta & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1-\delta \end{bmatrix} = P^T L U.$$

Note that this is equivalent to writing $PA = LU$ where P is another permutation (which undoes the action of P^T).

If we wish to control the multipliers, it's natural to choose the permutation P so that each of the multipliers is at most one. This standard choice leads to the following algorithm:

```
#
# Compute partial pivoted LU factors in packed storage
#
function mypivlu(A)
    n = size(A)[1]
    A = copy(A)
    p = Vector(1:n)
    for j = 1:n-1

        # Find pivot and do a swap
        _, jj = findmax(abs.(A[j:n,j]))
        jj += j-1
        for k = 1:n
            A[jj,k], A[j,k] = A[j,k], A[jj,k]
        end
        p[jj], p[j] = p[j], p[jj]
```

```
        # Compute multipliers and update Schur complement
        A[j+1:n,j] /= A[j,j]
        A[j+1:n,j+1:n] -= A[j+1:n,j]*A[j,j+1:n]'

    end
    p, UnitLowerTriangular(A), UpperTriangular(A)
end
```

In practice, we would typically use a strategy of *deferred updating*: that is, rather than applying the pivot immediately across all columns, we would only apply the pivoting within a block of columns. At the end of the block, we would apply all the pivots simultaneously. As with other blocking strategies we have discussed, this has no impact on the total amount of work done in some abstract machine model, but it is much more friendly to the memory architecture of real machines.

By design, this algorithm produces an L factor in which all the elements are bounded by one. But what about the U factor? There exist pathological matrices for which the elements of U grow exponentially with n. But these examples are extremely uncommon in practice, and so, in general, Gaussian elimination with partial pivoting does indeed have a small backward error. Of course, the pivot growth is something that we can monitor, so in the unlikely event that it *does* look like things are blowing up, we can tell there is a problem and try something different. But when problems do occur, it is more frequently the result of ill-conditioning in the problem than of pivot growth during the factorization.

Beyond partial pivoting

Gaussian elimination with partial pivoting has been the mainstay of linear system solving for many years. But the partial pivoting strategy is far from the end of the story! GECP, or Gaussian elimination with *complete* pivoting (involving both rows and columns), is often held up as the next step beyond partial pivoting, but this is really a strawman — though complete pivoting fixes the aesthetically unsatisfactory lack of backward stability in the partial pivoted variant, the cost of the GECP pivot search is more expensive than is usually worthwhile in practice. We instead briefly describe two other pivoting strategies that are generally useful: *rook pivoting* and *tournament pivoting*. Next week, we will also briefly mention *threshold pivoting*, which is relevant to sparse Gaussian elimination.

Rook pivoting

In Gaussian elimination with rook pivoting, we choose a pivot at each step by choosing the largest magnitude element in the first row *or column* of the current Schur complement. This

eliminates the possibility of exponential pivot growth that occurs in the partial pivoting strategy, but does not involve the cost of searching the entire Schur complement for a pivot (as occurs in the GECP case).

For the problem of solving linear systems, it is unclear whether rook pivoting really holds a practical edge over partial pivoting. The complexity is not really worse than partial pivoting, but there is more overhead (both in runtime and in implementation cost) to handle deferred pivoting for performance. Where rook pivoting has a great deal of potential is in Gaussian elimination on (nearly) *singular* matrices. If $A \in \mathbb{R}^{m \times n}$ has a large gap between σ_k and σ_{k+1} for $k < \min(m, n)$, then GERP on A tends to yield the factorization

$$PAQ = LU, U = \begin{bmatrix} U_{11} & U_1 2 \\ 0 & U_{22} \end{bmatrix}$$

where $U_{11} \in \mathbb{R}^{k \times k}$ and $\|U_{22}\|$ is very small (on the order of σ_{k+1}).

Rook pivoting and the closely-related threshold rook pivoting are particularly useful in constrained optimization problems in which constraints can become redundant. Apart from its speed, the rook pivoting strategy has the advantage over other rank-revealing factorizations that when A is sparse, as one can often control the fill (nonzeros in L and U that are not present in A). The LUSOL package of Michael Saunders is a particularly effective example.

Tournament pivoting

In parallel dense linear algebra libraries, a major disadvantage of partial pivoting is that the pivot search is a communication bottleneck, even with deferred pivoting. This is increasingly an issue, as communication between processors is far more expensive than arithmetic, and (depending on the matrix layout) GEPP requires communication each time a pivot is selected. For this reason, a number of recent *communication-avoiding* LU variants use an alternate pivoting strategy called *tournament pivoting*.

The idea behind *tournament pivoting* is to choose b pivot rows in one go, rather than iterating between choosing a pivot row and performing elimination. The algorithm involves each processor proposing several candidate pivot rows for a heirarchical tournament. There are different methods for managing this tournament, with different levels of complexity. One intriguing variant, for example, is the remarkable (though awkwardly named) `CALU_PRRP` algorithm, which uses rank-revealing QR factorizations to choose the pivots in the tournament. The `CALU_PRRP` algorithm does a modest amount of work beyond what is done by partial pivoting, but has better behavior both in terms of communication complexity and in terms of numerical stability.